Fault Mitigation of Switching Lattices under the Stuck-At-Fault Model

Lorena Anghel
Phelma Grenoble INP, TIMA Laboratory
Grenoble-Alpes University, France
lorena.anghel@imag.fr

Anna Bernasconi
Dipartimento di Informatica
Università di Pisa, Italy
anna.bernasconi@unipi.it

Valentina Ciriani
Dipartimento di Informatica
Università degli Studi di Milano, Italy
valentina.ciriani@unimi.it

Luca Frontini
INFN
Milano, Italy
luca.frontini@mi.infn.it

Gabriella Trucco
Dipartimento di Informatica
Università degli Studi di Milano, Italy
gabriella.trucco@unimi.it

Ioana Vatajelu
TIMA laboratory
Grenoble-Alpes University, France
ioana.vatajelu@univ-grenoble-alpes.fr

***Abstract*—Switching lattices are two-dimensional arrays composed by four-terminal switches (crossbar arrays). The idea of using regular two-dimensional arrays of switches to implement Boolean functions was proposed by Akers in 1972. Recently, with the advent of a variety of emerging nanoscale technologies, lattices have found a renewed interest. Switching lattices can have a non-negligible defective ratio. In this paper, we analyze the fault tolerance of switching lattices under the stuck-at-fault model (SAFM). We first identify the critical switches with a sensitivity analysis of the lattice. We then propose some techniques to improve the resilience to faults, which are implemented as a post preprocessing step after logic optimization.**

***Index Terms*—Switching lattices, Stuck-At-Fault Model, fault tolerance.**

I. Introduction

Recent years advancements in process scaling and 3D monolitical integration brought multiple possibilities for emerging devices to push further integration of electronic circuits. Nano-crossbars are among one of the most promising alternative solutions technology beyond CMOS devices [25]. They lead to programmable circuit architectures based on nano-crossbar arrays which operate similarly to conventional programmable logic arrays (PLAs), molecular switch crossbar arrays, and resistive crossbar logic [2], [12]. Due to simpler and cheaper manufacturing techniques, programmable crossbars result in regular and dense form [6] that are area and power efficient [2]. The computation is done on matrixes of switching elements that can be made of two-terminal switches i.e diodes [13], or resistive/memristive elements [19] or FET transistos [20]. Four terminal switches is also a great possibility particularly well adapted to dual logic functions [1], [16]. The memrisitve based cross points represents probably the most prominent solution explored by several research groups [14], adopted due to their potential to scale down to 5nm, compatibility to CMOS process, and also their potential to be used as memory elements. Indeed, due to the non-volatility of these devices, they offer possibilities for implementation of memory-intensive computing paradigms, enabling non Von-Neuman logic-in-memory operations [4]. This paradigm, allows logic and arithmetic operations to be directly processed in the memory within the crossbar array, making them attractive from neuromorphic applications such as prediction, classification and decision-making problems [9], [18]. Memristive devices can be programmed to store either two states (binary) or more than two (analog), when multiple resistance states are used together. These emerging technologies are quite immatures, prone to important defect densities (induces by spot defects, dust, assemblage faults, imperfections of the circuit), or instabilities that affect their yield. The faults that can be found in these technologies can be classified into two categories: soft faults and hard faults [8], [22]. Soft faults are caused by different cycle-to-cycle or device-to-device variations that appear during the fabrication, but also in-field during read/write operations [24]. Hard faults are provoked by fabrication steps or they can be caused by the forming process or by continuous stress; they are more difficult to be prevented. One typical type of hard fault occurs when the resistance of a resistive memory cell will no longer change; this category includes stuck-at-0 (SA0) and stuck-at-1 (SA1) faults caused by fabrication techniques and limited endurance. In this case, the faulty device is stuck at high resistance or low resistance state, and these situations are occurring with

Funded by the European Union Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 691178.

a quite high probability. It is reported that 63% of a storage array based on memristor is fault free in a 4Mb resistive RAM, with about 10% of the cells being of Stuck-At type [5]. In [23] the authors showed that 10% of broken memristor cells will lead to substantial degradation of the accuracy and overall performances of a convolutional neural network implemented on this structure. All these studies are performed on crossbar arrays for binary resistive devices. Very limited research has been dedicated to analyze faults model of resistive devices where multiple resistances are used. Since the fabrication technology of memristive cells is sensitive to different process steps (i.e., forming), it is very difficult to prevent SA faults during fabrication process [8], [5] which are therefore considered as predominant. Based on these above-mentioned fault models defined for memristors, various testing methods have been proposed in [7], [11], [15]. Testing provides a fault map that can be used for marking faulty devices and help reorganizing the programmability of the crossbar around defected cells through algorithm remapping [23]). This generally necessitates important area overheads, as sufficient spare cells organized on columns, lines or blocks have to be provided. Moreover, spare cells are usually considered as defect free, which is not always the case. In addition, as mentioned previously, variability of memristor resistances during write operations may also push the device in a hard Stuck-at fault. Understanding the impact of the sensitivity to defects and transient faults of memrisitive based devices on the mapping algorithm is a key step for future development. Yield analysis of nanocross-bars for uniformly and clustered distributed defects have been performed in [17], [21]. In this paper, our contributions are as follows: 1) We propose a fault injection method and tools in crossbar lattices which substitute a single cell with an always SA1 or SA0 cell. The fault injection algorithm uses uniform distribution. 2) The sensitivity of a decomposition algorithm on a given crossbar is analyzed face to SA0 and SA1. 3) The prior sensitivity analysis of help identifies critical switches. Further to that we propose mitigation factors to strengthen the mapping algorithm while keeping the crossbar area minimal.

The paper is organized as follows. Section II explain the logic function synthesis method on crossbar switching arrays. Section III discuss the fault model and the sensitivity analysis method. Section IV and V discuss methods for mitigation and finally Section VI presents sensitivity results.

II. Switching Lattices and Synthesis Methods

A switching lattice is a two-dimensional array of four-terminal switches. The four terminals of the switch link to the four neightbours of a lattice cell, so that these are either all connected (when the switch is ON), or disconnected (when the switch is OFF). A Boolean function can be implemented by a lattice in terms of connectivity across it:

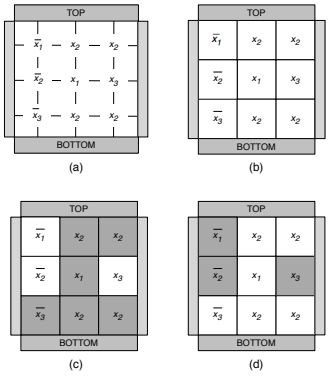

Fig. 1. A four terminal switching network implementing the function $f = \overline{x}_1\overline{x}_2\overline{x}_3 + x_1x_2 + x_2x_3$ (a); its corresponding lattice form (b); the lattice evaluated on the assignments 1,1,0 (c) and 0, 0, 1 (d), with grey and white squares representing ON and OFF switches, respectively.

- each four-terminal switch is controlled by a literal;
- if the literal takes the value 1, the corresponding switch is connected to its four neightbours, else it is not connected;
- the function evaluates to 1 if and only if there exists a connected path between two opposing edges of the lattice, i.e., the top and the bottom edges;
- input assignments that leave the edges unconnected correspond to output 0.

For instance, the network of switches in Figure 1 (a) corresponds to the lattice form depicted in Figure 1 (b), which implements the function $f = \overline{x}_1\overline{x}_2\overline{x}_3 + x_1x_2 + x_2x_3$. If we assign the values 1, 1, 0 to the variables x_1, x_2, x_3, respectively, we obtain paths of gray square connecting the top and the bottom edges of the lattices (Figure 1 (c)), on this assignment f evaluates to 1. On the contrary, the assignment $x_1 = 0, x_2 = 0, x_3 = 1$, on which f evaluates to 0, does not define any path from the top to the bottom edge (Figure 1 (d)).

The synthesis objective on a lattice consists in finding an assignment of literals to switches in order to implement a given target function with a lattice of minimal size. The size is measured in terms of the number of switches in the lattice.

A switching lattice can similarly be equipped with left edge to right edge connectivity, so that a single lattice can implement two different functions. This fact is explained in [3] where the authors propose a synthesis method for switching lattices simultaneously implementing a function f according to the connectivity between the top and the bottom plates, and its dual function f^D according to the connectivity between the left and the right plates. Recall that the dual of a Boolean function f depending on n binary variables is the function f^D such that $f(x_1, x_2, \ldots, x_n) = \overline{f^D}(\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n)$. This method produces lattices with a size that grows linearly with the number of products in an irredundant sum of product

(SOP) representation of f, and consists of the following steps:

1) find an irredundant, or a minimal, SOP representation for f and f^D: $SOP(f) = p_1 + p_2 + \cdots + p_s$ and $SOP(f^D) = q_1 + q_2 + \cdots + q_r$;
2) assign each product p_j $(1 \leq j \leq s)$ of $SOP(f)$ to a column and each product q_i $(1 \leq i \leq r)$ of $SOP(f^D)$ to a row;
3) for all $1 \leq i \leq r$ and all $1 \leq j \leq s$, assign to the switch on the lattice site (i, j) one literal which is shared by q_i and p_j (the fact that f and f^D are duals guarantees that such a shared literal exists for all i and j).

Note that, we can have a couple of products q_i and p_j such that the intersection of their literals is a set of cardinality greater that one. For building the corresponding lattice, the algorithm imposes to choose randomly one of the common literals. In this case, we denote the corresponding cell (i, j) as a *cell with multiple choice*. Moreover, note that, in Step 2 of the synthesis algorithm, the assignments of products to rows and columns is random. For example, consider the function f in ISOP form $f = x_1x_2 + x_1x_3 + x_2x_3$ and its dual $f^D = x_1x_2 + x_1x_3 + x_2x_3$. The lattice containing the cells with multiple choice is depicted in Figure 3. At the top of Figure 4, we have two possible lattices derived from the former by choosing a literal in the cells with multiple choice.

A second approach to the synthesis of minimal-sized lattices is proposed in [10], where the authors transform the synthesis problem into a satisfiability problem in quantified Boolean logic and solve it using a quantified Boolean formula solver.

III. Stuck-At-Faults in Lattices

The classical SAF model is well-known and classically used throughout the CMOS industry for many years. In CMOS, the SA model assumes that a defect causes a basic cell input or output to be fixed to either 0 or 1. Thus, all defects with this effect can be detected by tests for stuck-at-faultsi test. In a lattice, a SAF can be similarly modeled as a fixed value (0 or 1) in the faulty cell (i.e., a four-terminal switch) of the lattice.

In this section we briefly summarize the methodology described in [16] for the fault injection, which we exploit for the sensitivity analysis of our approach. The fault injectionis performed substituting a single cell with an always stuck-at 1 (SA1) or stack-at 0 (SA0) cell. The fault injection procedure is repeated for each cell of the lattice The simulation algorithm then generates all the 2^n possible inputs. For each input $x_1, \ldots, x_n$ the algorithm compares the given output with the correct one (i.e., $f(x_1, \ldots, x_n)$. Note that, we are not interesten in evalueting the sensitivity of the dual function f^D.

Let r and s be number of rows and columns, respectively, in a lattice. Let E^0_{ij} (resp., E^1_{ij}), with $1 \leq i \leq r$, $1 \leq j \leq s$, be the number of defective outputs with a SA0 (resp., SA1) in the cell (i, j) of the given lattice. When E^0_{ij} (resp., E^1_{ij}) is equal to 0 and, for any possible input, the lattice functional output is not affected by the SAF in the cell (i, j). In this case, the cell (i, j) is considered *robust* w.r.t. SA0 (resp., SA1). Let R^0 (resp., R^1) be the total number of robust cells w.r.t. SA0 (resp., SA1) in the lattice. Finally, let $E^0 = \sum_{i=1}^{i=r} \sum_{j=1}^{j=s} E^0_{ij}$ (resp., $E^1 = \sum_{i=1}^{i=r} \sum_{j=1}^{j=s} E^1_{ij}$) be the total number of defective outputs with SA0 (resp. SA1) in the simulation. Consider, for example, the function $f = x_4\overline{x}_5x_7 + \overline{x}_4x_6\overline{x}_7 + \overline{x}_4x_5\overline{x}_6x_7 + x_4\overline{x}_6\overline{x}_7 + x_4x_6x_7$ represented by the lattice in Figure 2 (a) (derived with the method in [3]). Figures 2 (b) and 2 (c) show the sensitivity map containing the value E^0_{ij} and E^1_{ij} in each cell. In order to evaluate the sensitivity of a lattice to SA0 and SA1 defects, we propose a metric that provides the average number of defective outputs: *Sensitivity of lattice* is the total number of inputs that propagetes a SA0/SA1 to an uncorrected output divided by the total number of inputs. In the case of SA0, $S^0_L = E^0/(2^n(r \times s))$, and for SA1, $S^1_L = E^1/(2^n(r \times s))$.

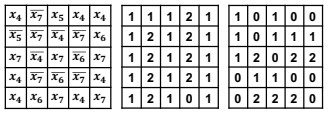

Fig. 2. a) Lattice design for the example function f and its sensitivity map for b) SA0 and c) SA1.

IV. Properties of lattice synthesized with the Altun-Riedel synthesis method

We now discuss some characteristics of the switching lattices obtained with the Altun-Riedel synthesis method [3] that could be exploited to enhance their fault tolerance.

First of all, we note that the Altun-Riedel method defines many equivalent lattices for the given function f. Indeed, in the second step of the procedure (see Section II) each product in an irredundant SOP for f is assigned to a column, and each product in an irredundant SOP for the dual f^D is assigned to a row, without any specific rule for these assignments. As a consequence, any permutation of the products in $SOP(f)$ and in $SOP(f^D)$ gives rise to a correct, and possibly different, lattice for f. Moreover, once each pair of products (one from $SOP(f)$ and one from $SOP(f^D)$) has been assigned to a lattice cell, the controlling literal is selected choosing arbitrarily one of the literals shared by both products. Thus, we can have multiple choices for all cells with multiple choice.

Taking into account these degrees of freedom, we now evaluate the number of potentially different lattices produced by this synthesis procedure. Suppose that $SOP(f)$ contains s products, and $SOP(f^D)$ contains r products. The lattice for f has dimension $r \times s$. Let us denote by $S(i, j)$ the subset of literals shared by the products assigned to the cell (i, j), and by $s_{i,j}$ the cardinality of this set. We have

Proposition 1: The number N_f of lattices for f produced by the Altun-Riedel method is given by $N_f = r!s! \prod_{\substack{1 \leq i \leq r \\ 1 \leq j \leq s}} s_{i,j}$.

Proof. Immediately follows since there are $r!$ ways to assign the products of $SOP(f^D)$ to the rows of the lattice, $s!$ ways

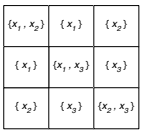

Fig. 3. A lattice for the function $f = x_1x_2 + x_1x_3 + x_2x_3$, with multiple choices on the diagonal cells.

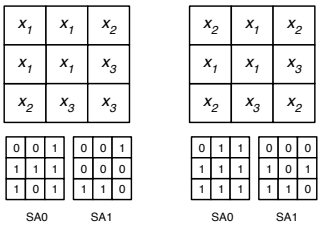

Fig. 4. Two lattices for $f = x_1x_2 + x_1x_3 + x_2x_3$ (see Figure 3 for the multiple choice lattice), with different sensitivity to SA0 and SA1 defects.

to assign the products of $SOP(f)$ to the columns, and $s_{i,j}$ ways to select the controlling literals of each cell (i, j). ■ Observe that N_f can be exponential in the lattice size:

Corollary 1: Let f depend on n binary variables. Then $N_f = O(r!s!n^{rs})$.

Proof. Easy follows as $s_{i,j} \leq n$. ■

Thus, the Altun-Riedel method provides many equivalent lattices for the same specified function f, all of dimension $r \times s$, and these lattices may exhibit a different SAF sensitivity for a single fault. Consider, for example, the lattice for $f = x_1x_2 + x_1x_3 + x_2x_3$ depicted in Figure 3, with cells with multiple choice on the diagonal. Starting from this lattice we can build up to $288 = 3!3!8$ lattices by permuting rows and columns and by choosing the controlling literal fo the diagonal cells. For instance, by simply making different choices at the diagonal cells, we could get the two lattices in Figure 4, which exhibit a different sensitivity to SA0 and SA1 defects: $S^0_L = 1/12$ and $S^1_L = 1/24$ for the first lattice, and $S^0_L = 1/9$ and $S^1_L = 1/18$ for the second one. In particular, the first lattice contains more robust cells, probably as a consequence of the many adjacent cells with the same controlling literal that might help to contain the effect of a faulty cell. Therefore, instead of picking a random permutation of the products in the starting SOPs, and selecting arbitrarily the controlling literal for all cells with multiple choice, one should exploit the degrees of freedom offered by the Altun-Riedel method to detect, among the N_f different lattices, the most resilient one. This issue will be discussed in the next Section V.

Finally, the possibility of permuting rows and columns is not guaranteed in lattices synthesized with other strategies (e.g., [10]). Consider for instance the 3×3 lattice L for the function $f = x_1x_2\overline{x}_3 + x_1\overline{x}_2x_3 + \overline{x}_1x_2x_3$ shown in Figure 5 (a). This lattice has not been derived with the Altun-Riedel method, that would instead produce a lattice of size 4×3. Permuting the columns of L, we can derive the lattice

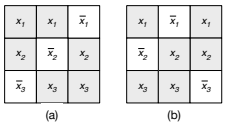

Fig. 5. A lattice L for $f = x_1x_2\overline{x}_3 + x_1\overline{x}_2x_3 + \overline{x}_1x_2x_3$ (a); a lattice obtained from L by a column permutation (b). Both lattices are evaluated on the assignment $x_1 = 1, x_2 = 1, x_3 = 1$.

in Figure 5 (b) that does not implement the function f, as it contains an accepting path for the off-set minterm $x_1x_2x_3$.

V. Characterization and Construction of the Most Resilient Lattice

Let us consider the Altun-Riedel synthesis method [3] and a Boolean function f. By the discussion in Section IV, we know that the different possible lattices for f, generated by the synthesis algorithm, are exponential in number. Thus, the main aim of this section is the study of efficient strategies to select the lattice that is less sensitive to cell defects.

Consider the two lattices shown in Figure 6. The two lattice are derived applying Altun-Riedel method to the function $f = x_1 + x_2x_4x_5 + x_3x_4x_5$. Let us assume a SA0 on the first cell on the top-left (depicted in gray in the lattices). While the lattice on the left computes a different function , i.e., $f' = x_1 + x_3x_4x_5$, the lattice on the right computes the correct function even in presence of the SAF. We can observe that the lattice on the right is derived from the first by a simple permutation of columns. In particular, in the second lattice, two similar columns are adjacent. The example gives us the intuition that, in order to decrease the sensitivity to cell defect, we should bring near cells containing the same literal. In fact the product that is no more computed by the faulty version of the first lattice (i.e., $x_2x_4x_5$), is computed by the second lattice using a "bypass" starting at the top of the second column going down and then on the left (i.e., path x_4, x_5, x_5, x_2). Note that this "bypass" is not possible in the lattice on the left since the two involved columns are not adjacent. Motivated by this observation, we describe several methods that try to keep adjacent cells containing the same controlling literals. We first give a metric that eases the description of the proposed techniques. Two cells in a lattice are *adjacent* if they are in the same column and in two adjacent rows or in the same row and in two adjacent columns. Consider a lattice L where each cell contains a controlling literal. For each cell c in L we define a_c the number of cells adjacent to c in L containing the same literal that is in c. The value a_L is $\sum_{c \in L} a_c$. In order to maximize the number of adjacent cells containing the same literal we must maximize a_L.

The synthesis algorithm by Altun-Riedel produces a lattice containing cells with multiple choices (e.g., the lattice shown in Figure 3). The approach we proposes is based on three

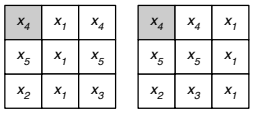

Fig. 6. Two equivalent lattices for the function $f = x_1 + x_2x_4x_5 + x_3x_4x_5$. While, in case of a SA0 in the first cell on top-left, the first lattice computes a different function, the second one still computes f.

algorithms, each starting with a lattice (containing cells with multiple choices) produced by Altun-Riedel's algorithm:

- *PermuteColumns*: make a random choice for the cells with multiple choice and permute the columns in order to maximize the number of adjacent cells containing the same literal (i.e., a_L).
- *PermuteRows*: make a random choice for the cells with multiple choice and permute the rows of a given lattice in order to maximize the number of adjacent cells containing the same literal (i.e., a_L).
- *ChooseLiteral*: given a lattice containing cells with multiple choice, in each cell with multiple choice chooses the literal that maximize the number of adjacent cells containing the same literal (i.e., a_L).

Note that the three algorithms return one of the N_f possible lattices produced by Altun-Riedel's algorithm. In other words, the proposed procedures make deterministic choices aiming at reducing the sensitiveness to defects, instead of the random choices performed by Altun-Riedel's algorithm.

VI. Experimental Evaluation

In this section we report some experimental results in order to analyze the fault sensitivity of switching lattices under the stuck-at-fault model. Our aim is to determine the strategy that allows to obtain the function decomposition less sensitive to SA0 and SA1 defects. For this purpose, for a given benchmark we consider five different lattices:

- L1: initial generic lattice, where no algorithms have been applied to maximize the number of adjacent cells containing the same literal (i.e., [3] method)
- L2: lattice obtained by the *PermuteRows* algorithm;
- L3: lattice obtained by the *PermuteColumns* algorithm;
- L4: lattice obtained by applying both *PermuteRows* and *PermuteColumns* algorithms in the same time;
- L5: lattice obtained by the *ChooseLiteral* algorithm.

To compute the best permutation of rows and columns we use the linear optimizer GLPK (GNU Linear Programming Kit). The simulation of GLPK on each input case is stopped after 1 hour in case the optimal solution is not computed. If the simulation is stopped after 1 hour (then without obtaining the optimal solution) GLPK produces a percentage giving a measure of how the obtained solution if far from the optimal one. In Table I, we mark these cases with the simbol '$\star$'.

Given a benchmark, after the computation of the 5 lattices, we inject errors and then we compute the metric described in Sect. III to evaluate the proposed strategies. The experiments have been run on a machine with two AMD Opteron 4274HE for a total of 16 CPUs at 2.5 GHz and 128 GByte of main memory, running Linux CentOS 7. The benchmarks functions are expressed in PLA form and are taken from a subset of LGSynth93 [26]. A total of about 620 functions were considered, and each output of a function is implemented as a separate Boolean function. The software used for simulations is written in C++. Since the simulation time depends on the number of variables, we consider lattices with a number of variables lower than 8. Note that this limitation is due to the onerous procedure for the fault simulation, and it is not due to our proposed algorithms.

In Table I we report the sensitivity of lattices to SA0 and SA1 defects. Due to lack of space, the reported values are a significant subset of the obtained one. The first column reports the name and the output number of the considered benchmark and lattice dimension ($r \times c$); the second column reports the number of variables. The following columns report, by group of two, the results for each computed lattice, by the metric described in Sect III, for SA0 and SA1 respectively.

Table II presents the percentages (*% more resilient lattices*) representing the amount of L2 to L5 lattices with higher resilience to faults w.r.t. the corresponding lattices in the set L1, and the percentages of average gains (*average gain*). To compute the percentages in Table II we consider the lattices in L2 (L3) with more than two rows (columns), and the lattices in L4 with at most two columns or at most two rows, and lattices in L5 that present cells with a multiple choice.

We may observe that all the proposed techniques allows to improve the resilience to faults. In particular, while for SA0 the best results are obtained with row permutation (i.e., L2), for SA1 the best results are obtained with columns permutation (i.e., L3). The permutation of rows and columns (i.e., L4) seems to be a good solution for both SA1 and SA0. The results obtained for the L5 set seem to be very encouraging.

VII. Conclusion

Emerging nanoscale technologies are very promising for circuit level implementation in cross-bar structures. However, due to their immature process, they can have non-negligible defect ratio and important variability. In this paper, we have proposed a method to analyze fault sensitivity of switching lattices under the stuck-at-fault model (SAF). Algorithmic improvements of the fault resilience has been proposed, exploiting different redundant schemes, such as literal selection, row or column permutations, or combination of both. Future work includes the study of soft errors, for which an error detection and correction phase should be deployed in addition to the remapping technique.

TABLE I
A COMPARISON OF S_L^0 AND S_C^0 BETWEEN LATTICES L1, L2, L3, L4 AND L5. BEST RESULTS ARE IN BOLD FACE.

			L1		L2		L3		L4		L5	
	$r \times s$	n	S_L^0	S_L^1	S_L^0	S_L^1	S_L^0	S_L^1	S_L^0	S_L^1	S_L^0	S_L^1
al2(0)	3×5	7	0.013	**0.133**	**0.011**	0.136	0.013	**0.133**	**0.011**	0.136	0.013	**0.133**
al2(13)	6×5	7	0.047	0.007	**0.039**	**0.003**	0.050	0.005	**0.039**	**0.003**	0.047	0.007
alcom(2)	2×4	5	0.070	**0.016**	**0.063**	0.023	0.070	**0.016**	**0.063**	0.023	0.070	**0.016**
alu2(2)	11×10	8	0.004	0.012	**0.002**	**0.005**	0.004	0.013	**0.002**	0.006	0.004	0.012
b9(0)	9×10	7	0.009	0.020	**0.007**	0.019	0.010	**0.014**	**0.007**	**0.014**	**0.007**	0.015
b11(3)	3×6	6	0.038	0.043	**0.028**	0.033	0.039	0.038	**0.028**	**0.026**	0.038	0.043
bench(7)	4×6	6	**0.317**	0.284	0.320	**0.281**	**0.317**	0.283	0.320	0.282	0.319	0.284
clpl(0)	4×4	7	**0.025**	0.092	0.092	**0.025**	**0.025**	**0.087**	**0.025**	**0.087**	**0.025**	0.092
dc2(3)	12×14	7	0.005	0.003	**0.002**⋆	0.003⋆	0.005⋆	**0.002**⋆	0.003⋆	0.003⋆	0.005	0.003
fout(9)	10×12	6	0.021	0.030	**0.005**⋆	0.021⋆	**0.005**	0.025	**0.005**	**0.016**⋆	0.006	0.019
in6(0)	2×7	8	0.021	0.034	0.020	0.036	0.021	0.034	0.020	0.036	**0.001**	**0.001**
luc(7)	4×7	6	0.009	0.040	**0.006**	0.032	0.009	0.036	**0.006**	**0.031**	0.009	0.040
luc(13)	9×10	6	0.006	0.012	**0.004**	0.011	0.006	0.010	**0.004**	**0.008**	**0.004**	0.015
pope.rom(7)	15×11	6	0.004	0.014	**0.002**	0.013	0.005⋆	0.010⋆	0.003	0.009⋆	0.003	**0.009**
rd53(2)	16×16	5	0.018	0.008	0.018⋆	0.004⋆	0.011⋆	0.015⋆	**0.008**⋆	0.004⋆	**0.008**	**0.0036**
t4(6)	3×3	4	0.118	**0.069**	**0.097**	0.083	0.118	**0.069**	0.104	0.076	0.118	**0.069**

TABLE II
COMPARISON OF PROPOSE ALGORITHMS WITH RESPECT TO L1.

	L2		L3		L4		L5	
	% more resilient lattices	average gain	% more resilient lattices	average gain	% more resilient lattices	average gain	% more resilient lattices	average gain
SA0	30%	17%	58%	20%	37%	27%	68%	22%
SA1	40%	25%	16%	20%	39%	25%	49%	22%

REFERENCES

[1] S. B. Akers, "A Rectangular Logic Array," *IEEE Trans. Comput.*, vol. C-21, no. 8, pp. 848–857, Aug 1972.

[2] D. Alexandrescu, M. Altun, L. Anghel, A. Bernasconi, V. Ciriani, L. Frontini, and M. Tahoori, "Logic synthesis and testing techniques for switching nano-crossbar arrays," *Microprocessors and Microsystems*, vol. 54, pp. 14–25, 2017.

[3] M. Altun and M. D. Riedel, "Logic Synthesis for Switching Lattices," vol. 61, no. 11, pp. 1588–1600, Nov 2012.

[4] J. Borghetti, G. S Snider, P. J Kuekes, J. J. Yang, D. Stewart, and S. Williams, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, 04 2010.

[5] C. Chen, H. Shih, C. Wu, C. Lin, P. Chiu, S. Sheu, and F. T. Chen, "Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme," *IEEE Transactions on Computers*, 2014.

[6] Y. Chen, G.-Y. Jung, D. A. A. Ohlberg, X. Li, D. R. Stewart, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, and R. S. Williams, "Nanoscale molecular-switch crossbar circuits," *Nanotechnology*, vol. 14, 2003.

[7] Y. Chen and J. Li, "Fault modeling and testing of 1t1r memristor memories," in *VTS*. IEEE Computer Society, 2015.

[8] R. Degraeve, A. Fantini, N. Raghavan, L. Goux, S. Clima, B. Govoreanu, A. Belmonte, D. Linten, and M. Jurczak, "Causes and consequences of the stochastic aspect of filamentary rram," *Microelectronic Engineering*, vol. 147, 2015.

[9] Y. Deng, P. Huang, B. Chen, X. Yang, B. Gao, J. Wang, L. Zeng, G. Du, J. Kang, and X. yan Liu, "Rram crossbar array with cell selection device: A device and circuit interaction study," *IEEE Transactions on Electron Devices*, vol. 60, 2013.

[10] G. Gange, H. Søndergaard, and P. J. Stuckey, "Synthesizing Optimal Switching Lattices," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, no. 1, pp. 6:1–6:14, Nov. 2014. [Online]. Available: http://doi.acm.org/10.1145/2661632

[11] S. Hamdioui, M. Taouil, and N. Z. Haron, "Testing open defects in memristor-based memories," *IEEE Transactions on Computers*, vol. 64, no. 1, 2015.

[12] M. Haselman and S. Hauck, "The Future of Integrated Circuits: A Survey of Nanoelectronics," *Proceedings of the IEEE*, vol. 98, no. 1, 2010.

[13] Y. Huang, X. Duan, Y. Cui, L. J. Lauhon, K.-H. Kim, and C. M. Lieber, "Logic gates and computation from assembled nanowire building blocks," *Science*, vol. 294, no. 5545, pp. 1313–1317, 2001.

[14] ITRS, "The International Technology Roadmap for Semiconductors," in *ITRS 2011 Edition*, 2011.

[15] S. Kannan, R. Karri, and O. Sinanoglu, "Sneak path testing and fault modeling for multilevel memristor-based memories," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013.

[16] M. C. Morgül, L. Frontini, O. Tunali, E. I. Vatajelu, V. Ciriani, L. Anghel, C. A. Moritz, M. R. Stan, D. Alexandrescu, and M. Altun, "Integrated Synthesis Methodology for Crossbar Arrays," in *IEEE/ACM International Symposium on Nanoscale Architectures*, 2018.

[17] H. Naeimi and A. DeHon, "A greedy algorithm for tolerating defective crosspoints in nanopla design," in *FPT*. IEEE, 2004.

[18] L. Ni, H. Huang, Z. Liu, R. V. Joshi, and H. Yu, "Distributed in-memory computing on binary rram crossbar," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, 2017.

[19] G. Snider, "Computing with hysteretic resistor crossbars," *Applied Physics A*, vol. 80, no. 6, pp. 1165–1172, 2005.

[20] G. Snider, P. Kuekes, T. Hogg, and R. S. Williams, "Nanoelectronic architectures," *Applied Physics A*, vol. 80, no. 6, pp. 1183–1195, 2005.

[21] Y. Su and W. Rao, "Defect-tolerant logic mapping on nanoscale crossbar architectures and yield analysis," in *2009 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2009.

[22] E. I. Vatajelu, P. Prinetto, M. Taouil, and S. Hamdioui, "Challenges and solutions in emerging memory testing," *IEEE Transactions on Emerging Topics in Computing*, 2018.

[23] L. Xia, M. Liu, X. Ning, K. Chakrabarty, and Y. Wang, "Fault-tolerant training with on-line fault detection for rram-based neural computing systems," in *54th Design Automation Conference (DAC)*, 2017.

[24] L. Xia, P. Gu, B. Li, T. Tang, X. Yin, W. Huangfu, S. Yu, Y. Cao, Y. Wang, and H. Yang, "Technological exploration of rram crossbar array for matrix-vector multiplication," *Journal of Computer Science and Technology*, vol. 31, no. 1, 2016.

[25] H. Yan, H. S. Choe, S. Nam, Y. Hu, S. Das, J. F. Klemic, J. C. Ellenbogen, and C. M. Lieber, "Programmable nanowire circuits for nanoprocessors," *Nature*, vol. 470, no. 7333, 2011.

[26] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0," Microelectronic Center, User Guide, 1991.